Safe Cryptography for All: Towards Visual Metaphor Driven Cryptography Building Blocks

Dirk van der Linden
University of Bristol
Bristol, UK
dirk.vanderlinden@bristol.ac.uk

Awais Rashid
University of Bristol
Bristol, UK
awais.rashid@bristol.ac.uk

Emma Williams
University of Bristol
Bristol, UK
emma.williams@bristol.ac.uk

Bogdan Warinschi
University of Bristol
Bristol, UK
csxbw@bristol.ac.uk

ABSTRACT

In this vision paper, we focus on a key aspect of the modern software developer's potential to write secure software: their (lack of) success in securely using cryptography APIs. In particular, we note that most ongoing research tends to focus on identifying concrete problems software developers experience, and providing workable solutions, but that such solutions still require developers to identify the appropriate API calls to make and, worse, to be familiar with and configure sometimes obscure parameters of such calls. In contrast, we envision identifying and employing targeted visual metaphors to allow developers to simply select the most appropriate cryptographic functionality they need.

KEYWORDS

cryptography, secure code, visual metaphor, developer support

ACM Reference Format:
Dirk van der Linden, Awais Rashid, Emma Williams, and Bogdan Warinschi. 2018. Safe Cryptography for All: Towards Visual Metaphor Driven Cryptography Building Blocks. In *SEAD'18: SEAD'18:IEEE/ACM 1st International Workshop on Security Awareness from Design to Deployment , May 27, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, Article 4, 4 pages. https://doi.org/10.1145/3194707.3194709

1 INTRODUCTION

Developing software is no longer the domain of the select few with deep technical skills, training and knowledge. Mobile and web app development and easy-to-program hardware devices, such as Arduino and Raspberry Pi, have resulted in a wide range of people from diverse backgrounds developing software. This diversity of developers is at the heart of a range of innovations that drive our digital economy. The software they produce can be, and is, deployed across systems pervasive in many aspects of human activity and is used by a global user base. But such software development by all holds several security implications.

Research has shown that use of cryptography APIs are a common cause of software vulnerability [4, 5], particularly in the mobile domain. Consider, for example, a basic client-server application where the client delegates to the server some computational task and where it is desired that the communication relating to this computation stays secret (from third parties). An inexperienced designer would conclude that it suffices to simply encrypt the client-server communication. There are however several other crucial yet non-obvious questions that need to be answered before concluding that encryption is sufficient. For example, the client needs to guarantee the identity of the server before it makes any query, encryptions should be authenticated [1], replay of messages that were sent encrypted should not be possible, et cetera.

Employing protocols (rather than vanilla encryption) goes a long way towards avoiding many subtle pitfalls. For example, when used with care, a secure channel protocol like the Transport Layer Security (TLS) takes care of entity and message authentication, integrity of the communication and so on. Furthermore, good, widely vetted protocol implementations are readily available through libraries. However, a developer using these libraries is faced with further challenges. The APIs of these libraries are complex and unintuitive, require users to set often obscure parameters, and select between many possible configurations – some secure, some insecure! Many attacks are in fact due to incompetent uses of existing libraries [5].

Shuai *et al.* investigated the misuse of cryptography APIs in Android applications, finding significant misuse, varying from the use of ECB-mode for encryption, use of broken algorithms such as DES, and having reversible one-way hashes [13]. Chatzikonstantinou *et al.* similarly analyzed 49 Android applications and found 87.8% to contain some degree of cryptographic misuse [2]. They proposed strategies for mitigating this misuse, by formulating guidelines and best practices for developers. However, such guidelines still place the stress of getting the code right on the individual developer, lacking e.g., tool support that generates the code for them, or at least nudges them towards correct use of the cryptography API.

To understand *why* developers have such trouble using cryptography APIs correctly, Nadi et al. [10] performed several empirical studies, focused on eliciting the difficulties developers face. They found that the key challenges in using a cryptography API correctly

SEAD 2018, May – June 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5727-2/18/05…$15.00
https://doi.org/10.1145/3194707.3194709

were identification of the correct sequence of method calls, understanding of the underlying API, and identification of the parameters to use. However, as these studies were based on self-reported data, to what extent developers actually understood the concepts correctly remains unknown.

Given the difficulty that developers have using cryptography APIs, while claiming to understand it, we believe this calls for support strategies that do not place the stress of correctly using such APIs on the developer. In order to ensure that such tools and guidance are as effective as possible in encouraging correct cryptographic API use, it is crucial that an in-depth understanding of how users understand this activity is developed. This must include both their understanding of the interface and how it operates, and their wider (potentially, lack of) understanding of the fundamental cryptography concepts on which it is based. Only once this basic research is conducted will it be possible to ensure that decision support tools are appropriately matched to the difficulties experienced by those using them. Rather than simply assisting the user in completing a task, this will allow for the development of adaptive mechanisms that aid future learning and understanding within user populations. To summarize the problem,

(1) cryptography functionality is vital for most modern software, but the use of cryptography APIs is challenging and has led to many security vulnerabilities
(2) current supporting strategies for developers are based on guidelines that tell developers what (not) to do while writing code, instead of taking the problem out of their hands

In the rest of this vision paper we set out a first step towards the longer-term goal of matching support to the user. In particular, we outline how the combination of using empirically grounded *visual metaphors* appropriate to specific communities of developers could side-step the need for them to understand cryptography in detail, and more importantly: allow a trade-off between flexibility of use and not allowing them to get things wrong. Section 2 details the developers we focus on: the young and inexperienced developers ruling the day in mobile software development. Section 3 discusses the kind of assistance available to them, followed by Section 4 which sets out a research agenda for eliciting metaphors for key cryptography functionality, and how this may aid in the design of tools supporting the use of cryptography APIs. Finally, we conclude in Section 5.

2 THE COMMUNITY: EVER YOUNGER AND LESS EXPERIENCED

A 2012 study of mobile software developers found that 40% worked as independents, 27% in 2–3 person organizations, and only 19% in organizations with > 10 people [3]. Recent professional reports [14] show that mobile developers have less experience, and are not as motivated by financial gains as others. For example, the solo 'hobbyists' (15%) who just want to have fun, and 'explorers' (28%) out learning and testing the market, constitute almost half of the developer demographic in 2016. Even outside of the inexperienced, other demographics constitute a rather significant further share of the population, such as 'hunters' (21%): independent developers focused on the money, and 'guns for hire' (15%): independent developers working on commission. With 52% of mobile developers having a revenue below $500 per month, financial success is not guaranteed.

Software development in the mobile domain thus seems to have become the playground of the young, independent, and inexperienced —and this is unlikely to change soon. Furthermore, of the (independent) developers working with established organizations, $\frac{1}{3}$ introduce personally acquired tools to those organizations. Young and/or independent developers thus seem to increasingly rule the day in mobile software development, meaning that focusing on security behaviors of individual developers has the potential to not only impact their own practices but also that of the organizations that may employ them.

Given the prevalence of young, inexperienced developers, and the need to stimulate them to produce secure software, understanding the way they think and conceptualize cryptography is of vital importance. API developers may be the typical 'gurus', older, more experienced and likely well-educated in relation to their technical background, which can lead to a significant conceptual misalignment between them and the developers.

3 THE SUPPORT: FROM TELLING TO SHOWING

As briefly mentioned before, developers have some support available to them to guide them towards correct use of cryptography APIs. However, much of this support takes the form of well-intended guidelines and best practices that do little to alleviate the key problem: the community of young and inexperienced developers simply does not 'get' the minutiae of using cryptography APIs to the same extent as the API developers (and well-meaning researchers) do [2, 10, 13].

For example, a recently proposed tool, CogniCrypt [9], makes a move in the right direction by providing code generation that securely uses certain cryptography functions and live static analyses of code as it is being written. However, the way the latter is implemented to support developers is based on normative statements that tell users that what they did was wrong, and what they should do instead, without necessarily explaining why or ensuring they will get it right. For example, a warning provided by CogniCrypt is:

> "The insecure algorithm DES is used. Please use secure algorithm (e.g. AES) instead."

Although providing such a warning may alert users to the fact that they have made an error, it does not explain why their choice is incorrect in this instance. As a result, the developer's understanding remains constrained, while still having to ensure that they use the API correctly. Instead, future iterations could take the responsibility of even writing the code that calls the API out of the hands of developers.

Over the past decade, a rather appropriate paradigm has become more popular, especially to teach younger developers: visual programming languages. Projects such as Scratch [12] and code.org [8] have shown the feasibility of creating programming environments designed around the interfaces that the modern developer community is used to. Touch Develop [15], for example, was designed with the idea of using only touchscreen input rather than the traditional

keyboard. A visual programming environment specifically designed for the creation of Android applications, App Inventor [17], has had significant uptake, and has been used to, e.g., introduce computing science concepts [6], and study how developers' general and domain-specific skill progresses as they develop more apps [18].

It is in this shift from *telling* to *showing* where we envision the safe use of cryptography APIs to be. In a drag and drop environment, developers could simply pick the functionality they want, and be assured of a correct and secure use. Instead of trying to understand documentation written by the API developer, they simply determine they want to send a message securely from sender to recipient, and drag the relevant blocks in place. With code generation as demonstrated by CogniCrypt [9] this would be feasible, but require us to make sure users understand exactly what they are dragging into their program. To ensure that, we need to understand how these developers think, and elicit appropriate visual metaphors that will allow for clear matching between what the developers wants to achieve, and what the cryptography functionality offers.

4 THE RESEARCH AGENDA: TOWARDS EMPIRICALLY GROUNDED APPROPRIATE CRYPTOGRAPHY METAPHORS

To provide visual building blocks that clearly indicate their underlying functionality, we need to understand how developers think. Language, whether visual or textual, and its underlying concepts is the language of community, and should be understood in the context of that community's habits and way of thought. [11]. Thus, to determine what metaphors are appropriate to use for a given concept, we have to engage with the community described in Sec. 2.

While visual metaphors are an essential part of conveying complicated concepts to lay users, even the common visual metaphors in cryptography are not necessarily as appropriate as they could be. Even in their seminal paper Why Johnny Can't Encrypt, Whitten and Tygar [16] noted issues with core visual metaphors, such as requiring more granularity in the notion of keys, and not using outdated symbolism such as quill pens for signatures. Given that our community is ever younger, using such outdated visual metaphors would be a problem indeed.

Instead of focusing on visual metaphors for individual concepts (e.g., quill for the concept of signature), as a first step we propose to elicit community-specific visual metaphors for essential cryptography *functionality*. From Nadi *et al.* we infer that essential cryptography functionality needed by developers is:

(1) securing connections and communications
(2) authenticating users (logins)
(3) encrypting files

4.1 Materials

To elicit visual metaphors from developers, we need to ensure they conceive of the functionality as objectively as possible. Thus, we need to avoid the abstract terminology such as 'authenticating' and 'encrypting', instead opting for clear descriptions of *what* is done in such functionality, free from jargon as much as possible. Some non-exhaustive examples follow below.

This may, for example, be done by describing abstract 'objects' of functionality, e.g., **something that secures connections and communications** as performing a sequence of actions:

- it makes sure you know exactly who the sender of the message is (optional)
- it makes sure you know exactly who is the recipient of the message
- it makes sure you know exactly what is/are the message(s) to be sent
- it makes sure nobody but the sender and recipient can read the message

Something that allows authentication of user(s) login(s) may be described as performing this sequence of actions:

- there are users with some identification
- those users have some password
- it only allows access to those users that know their passwords

Something that encrypts files may be described as performing this sequence of actions:

- it makes sure its contents are hidden
- it makes sure it is stored safely and locally

4.2 Method

4.2.1 Visual metaphor generation. Having a way to describe the core functionality free of jargon (and thus likely to avoid misinterpretation based on limited or wrong technical understanding), we need to generate visual metaphors. Participants should therefore not be constrained to text only. A study employing, e.g., participatory design of metaphors, would include for each of the three core functionalities a key question: *How do you conceive of an object that does the following?*, followed by a listing of functionality per the above. Participants would then be stimulated to co-design the metaphors, drawing any visual things that come to mind. Moreover, participants would be stimulated to verbalize their thoughts (in line with the think-aloud protocol), to allow for coding and analysis of their design thinking.

4.2.2 Coding of data. To analyze recorded verbal responses during the design session(s), data will be transcribed and subsequently coded in several iterations by multiple independent coders. First, each coder will perform two to three iterations grouping responses together. The resulting groupings between coders will be compared and used to derive codes that the textual answers may then be classified by.

To analyze the elicited visual responses, data will be coded in a similar way, with multiple independent coders grouping produced visual answers. The grouping of these visual answers will then be compared to the result of the textual response analysis, in order to provide a mapping between potentially matching textual and visual descriptions. This mapping can then be used to aid in the generation of final visual metaphors by taking the most re-occurring visual variables (e.g., shape, color, texture), as well as higher level visual information like coded symbolism.

4.3 Participants

A set of focused participatory design sessions generating visual metaphors is a first step in building targeted metaphors. Such sessions can involve carefully selected participants from different backgrounds with interests in coding.

Acknowledging the wide spectrum of of developers in the community described in Section 2, with distinct personal and professional make-up, performing the above again as a wider crowdsourced study would be appropriate to ensure information saturation – quite in line with elicitation of requirements via the crowd [7]. This could also shift the used method to e.g., carefully designed interactive surveys, so as to target a wider audience. We would first target specifically developers releasing software for the Android ecosystem, so as to delineate the community more effectively.

This community, however, should not be *a priori* considered to be homogeneous in their habits or attitudes, quite far from it. Through the coding of responses likely several clusters will arise of distinct conceptualization patterns which may, together with elicited demographic data, lead to the identification of smaller sub communities that conceptualize and work in particular ways. This can be used to produce essentially visual dialects of proposed metaphors, in order to offer personalized support for different groups.

5 CONCLUDING OUTLOOK

This vision paper sets out our vision towards secure cryptography API use by young and inexperienced developers, by moving them away from the actual source-code. Instead, we propose to follow the visual programming language paradigm, and use targeted empirically grounded visual metaphors to offer building blocks for the most needed cryptography functionality.

Our future work is set to implement the study described here, connecting eventually-produced personalized visual blocks with automatically generated code that uses key functionality, using e.g., CogniCrypt's existing generation of source-code correctly using cryptography APIs.

Acknowledgments. This work is partially supported by EPSRC grant EP/P011799/1, Why Johnny doesn't write secure software? Secure software development by the masses, and by a Vice-Chancellor's Fellowship awarded to the third author by the University of Bristol.

REFERENCES

[1] Mihir Bellare and Chanathip Namprempre. 2000. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 531–545.

[2] Alexia Chatzikonstantinou, Christoforos Ntantogian, Georgios Karopoulos, and Christos Xenakis. 2016. Evaluation of cryptography usage in android applications. In *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (formerly BIONETICS)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 83–90.

[3] Amy Cravens. 2012. A demographic and business model analysis of today's app developer. *GigaOM Pro* (2012).

[4] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 73–84.

[5] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. 2012. The most dangerous code in the world: validating SSL certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 38–49.

[6] Jeff Gray, Hal Abelson, David Wolber, and Michelle Friend. 2012. Teaching CS Principles with App Inventor. In *Proceedings of the 50th Annual Southeast Regional Conference (ACM-SE '12)*. ACM, New York, NY, USA, 405–406. https://doi.org/10.1145/2184512.2184628

[7] E. C. Groen, N. Seyff, R. Ali, F. Dalpiaz, J. Doerr, E. Guzman, M. Hosseini, J. Marco, M. Oriol, A. Perini, and M. Stade. 2017. The Crowd in Requirements Engineering: The Landscape and Challenges. *IEEE Software* 34, 2 (Mar 2017), 44–52. https://doi.org/10.1109/MS.2017.33

[8] Filiz Kalelioğlu. 2015. A new way of teaching programming skills to K-12 students: Code. org. *Computers in Human Behavior* 52 (2015), 200–210.

[9] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler, and R. Kamath. 2017. CogniCrypt: Supporting developers in using cryptography. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 931–936. https://doi.org/10.1109/ASE.2017.8115707

[10] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. Jumping Through Hoops: Why Do Java Developers Struggle with Cryptography APIs?. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 935–946. https://doi.org/10.1145/2884781.2884790

[11] Chaim Perelman. 1971. The new rhetoric. In *Pragmatics of natural languages*. Springer, 145–149.

[12] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67.

[13] Shao Shuai, Dong Guowei, Guo Tao, Yang Tianchang, and Shi Chenjie. 2014. Modelling analysis and auto-detection of cryptographic misuse in android applications. In *Dependable, Autonomic and Secure Computing (DASC), 2014 IEEE 12th International Conference on*. IEEE, 75–80.

[14] SlashData Developer Economics. 2017. Developer Economics: State of the Developer Nation. https://www.developereconomics.com/reports. (2017). Online; accessed 20 October 2017.

[15] Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, and Manuel Fahndrich. 2011. TouchDevelop: programming cloud-connected mobile devices via touchscreen. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*. ACM, 49–60.

[16] Alma Whitten and J Doug Tygar. 1999. Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0.. In *USENIX Security Symposium*, Vol. 348.

[17] David Wolber, Harold Abelson, and Mark Friedman. 2015. Democratizing Computing with App Inventor. *GetMobile: Mobile Comp. and Comm.* 18, 4 (Jan. 2015), 53–58. https://doi.org/10.1145/2721914.2721935

[18] Benjamin Xie and Hal Abelson. 2016. Skill progression in MIT app inventor. In *Visual Languages and Human-Centric Computing (VL/HCC), 2016 IEEE Symposium on*. IEEE, 213–217.